1 Introducción

En la actualidad, las computadoras y dispositivos electrónicos nos rodean. Desde las clásicas computadoras hasta televisiones con microprocesadores incluidos. Todos estos dispositivos requieren de herramientas de software para facilitar su uso y desarrollo de aplicaciones. Los sistemas operativos proporcionan las herramientas mas básicas de software para estas tareas. Existen diferentes sistemas operativos que varían en su utilidad, herramientas o arquitectura. Estos sistemas operativos son una parte esencial para la computación. Sin estos, los avances en computación no hubieran sido posibles. La pregunta por contestar es cómo se define a un sistema operativo. De acuerdo con Andrew S. Tannenbaum en su libro Sistemas Operativos Modernos: ``Es difícil puntualizar que es un sistema operativo''. Sin embargo, podemos decir que un sistema operativo contiene herramientas esenciales orientadas a facilitar la comunicación entre el hardware y el usuario. Cabe mencionar que los usuarios a los que haremos referencia y a los que están orientados estas herramientas es principalmente a desarrolladores y usuarios avanzados o power users. En adelante, la discusión del desarrollo de sistemas operativos se centrará en la manera de implementar estos servicios básicos, dejando de lado temas como el ambiente de ventanas o interfaces dirigidas al consumidor común.

1.1 Computer Architecture Overview

Para poder diseñar y por consecuencia entender como funcionan los sistemas operativos, es necesario entender el concepto detrás de las funcionalidades de los microprocesadores actuales. El poder entender el concepto abstracto detras de la concepción de las computadoras es esencial para entender la manera en que estas fueron diseñadas y por consecuencia las necesidades que debe cubrir un sistema operativo.

Todos los dispositivos actuales que hacen uso de microprocesadores están basados en una construcción matemática. Alan Turin definió en 1936 lo que llegaría a conocerse como Máquina de Turing (MT). Una MT consiste en una cinta con símbolos, una cabeza lectora y una máquina de estados. En la figura 1 se muestra un diagrama de una MT.

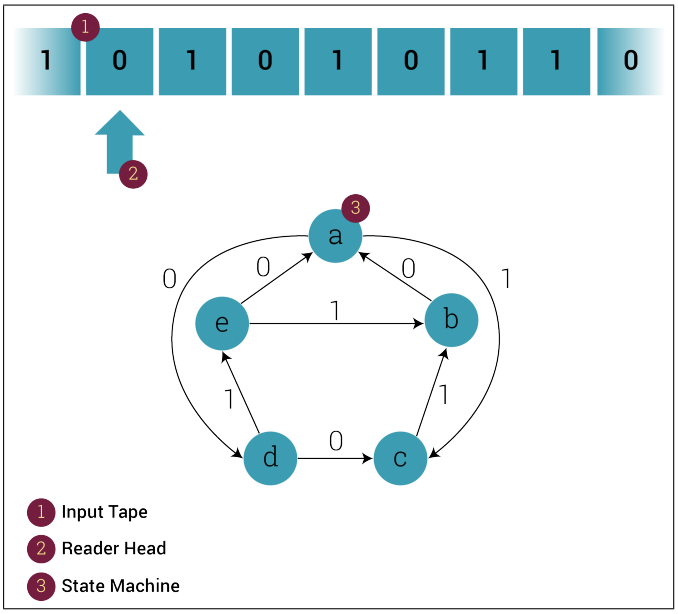


Figura 1: Representación Gráfica de una Turing Machine

La operación de una MT es como sigue: La cabeza lectora avanza un lugar, lee un símbolo de la cinta y la máquina de estados cambia su estado actual según el símbolo leído. Los microprocesadores modernos siguen el mismo principio. La memoria de acceso aleatorio (RAM por sus siglas en inglés) toma el lugar de la cinta, la cabeza lectora es un elemento del procesador conocido como program counter (PC) y por último la máquina de estados es el conjunto de elementos del procesador llamados registros, como `AX` o `DS`.

1.2 Tasks of an Operating System

Las tareas específicas de un sistema operativo dependen en su mayoría del objetivo con el que fue construido y el hardware donde es im-

plementado. Sin embargo, podemos generalizar sus tareas en dos elementos principales:

1. Proveer de herramientas y programas a desarrolladores y usuarios que sean sencillas y transparentes a diferencia de lo que seria operar el hardware de manera directa.

2. Administrar los diferentes elementos de hardware.

Esto significa que el sistema operativo sera empacado con un conjunto de librerias listas para ser usadas por los desarrolladores además de mantener servicios en ejecución durante el funcionamiento de la computadora en espera de brindar los servicios requeridos al usuario. Estos servicios son usados tanto por desarrolladores como por usuarios comerciales, sin embargo, estos últimos solo los usan de manera indirecta a través de herramientas más sofisticadas. Un sistema operativo actúa como una capa intermedia entre el hardware y los desarrolladores y usuarios para presentar una interfaz limpia, sencilla y ordenada de lo que de otra forma sería un trabajo solo reservado para personas entrenadas como lo era en periodos previos a la revolución de los sistemas operativos. Existe sistemas operativos para diferentes tareas. Desde sistemas operativos para los main frame que ocupan cuartos enteros hasta sistemas operativos para dispositivos móviles. Pasando por computadoras personales, clusters, computadoras multi núcleo y multi procesador, servidores, etc. La maquinaria industrial también requiere de sistemas operativos especializados como aquellos utilizados para operar conjuntos de sensores en fábricas o sistemas de tiempo real como los que operan los aviones Boeing.

Específicamente, podemos hablar de tres elementos básicos en las tareas de un sistema operativo: administración de la memoria, administración del tiempo de procesador y el sistema de mensajes o IPC por las siglas del nombre en inglés Inter-process Comunication.

El sistema de administración de memoria se encarga de controlar el acceso y otorgar espacios de memoria a los procesos en ejecución solicitantes. El sistema de administración de tiempo o scheduling se encarga de vigilar y otorgar el uso del procesador a los diferentes procesos que se encuentren en la cola de ejecución. Por último, el IPC es un sistema de mensajes para la comunicación entre procesos, es decir, gracias aeste servicio, los procesos en ejecución pueden comunicarse entre sí. Cabe destacar que no hay que confundir el IPC con el sistema de entrada salida (I/O). El sistema de I/O también es un servicio otorgado por el sistema operativo pero es una comunicación realizada en un nivel de abstracción más alto que el IPC.

1.3 Operating System Architecture

Uno de los elementos mas importantes y discutidos en la construccción de un sistema operativo es el kernel. La mayor parte de la lliteratura para el diseño de los sistemas operativos se centra en algoritmos y propuestas de diseño para el desarrollo de kernels. Otros servicios que también son implementados en los sistemas operativos empacados para el usuario final han sido catalogados en problemas separados al de la implementación de un sistema operativo.

El kernel o núcleo de un sistema operativo incorpora todas las funciones de bajo nivel requeridas para otorgar al usuario un ambiente con el que pueda empezar a operar. Note que el objetivo del kernel es otorgar funciones para la administracion de los recursos de hardware y no el de crear un ambiente completo de trabajo como es el caso de los gestores de ventanas.

Actualmente existen cuatro tipos de diseños enfocados al kernel de un sistema operativo: Monolithic Kernel, Microkernel, Nanokernel y Exokernel. Antes de continuar con la descripción de cada kernel, es necesario introducir El concepto de Ring. Ring es un concepto de seguridad en el que se definen capas de seguridad o anillos como se muestra en la figura 2.

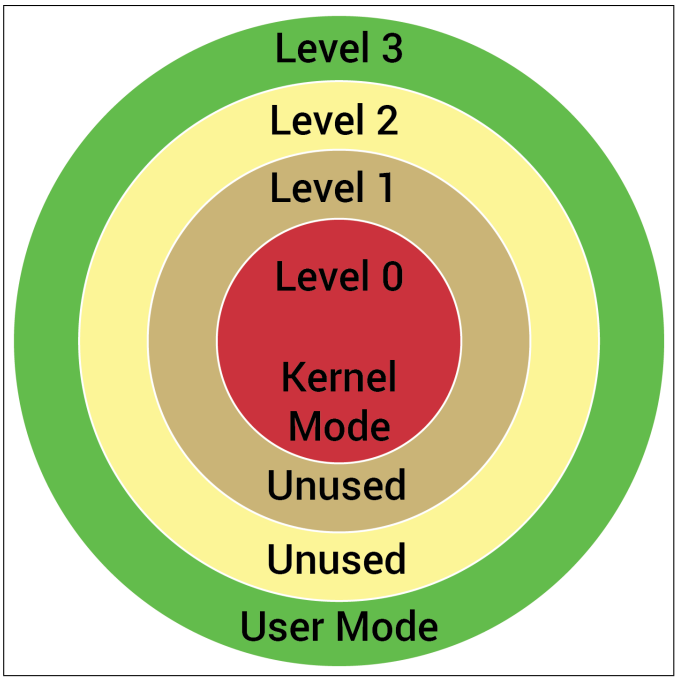


Figura 2: Anillos de Seguridad en al Arquitectura x86

Los servicios de un sistema operativo son ejecutados dentro de un anillo específico y no tiene acceso a modificar los servicios de capas inferiores de manera arbitraria. Este esquema de seguridad es soportado a nivel de hardware. En particular, las memorias comerciales para computadoras personales soportan dos anillos. A estos dos anillos se les conoce como kernel mode y user mode. La intención del kernel mode también conocido como nivel 0 es la de ejecutar todos los servicios críticos del kernel dentro de este nivel, manteniendo los servicios del usuario dentro del user mode o nivel 3. Esto evita que un usuario tenga acceso al hardware de manera directa o modifique el estado de un servicio inadvertidamente, o intencionalmente en el caso de virus de computadora o usuarios maliciosos. Otra de las razones de porque usar estos dos niveles a nivel de hardware es para implementar el concepto de multiusuario. En un sistema multiusuariolos usuarios comparten recursos de hardware. Si un usuario puede hacerse con el control de uno de estos recursos el sistema operativo se vería impedido de administrarlo y compartirlo con otros usuarios conectados al sistema. Todos los servicios considerados críticos se ejecutan en kernel mode. Cuales de los servicios son críticos varía de acuerdo al tipo de arquitectura. A continuación se enumeran los distintos tipos de arquitecturas y como sus servicios son clasificados.

1. Monolithic Kernel - El Monolithic Kernel es quizá el mas secillo en concepto. Todos los servicios de bajo nivel que administran el hardware se encuentran contenidos en un solo módulo. Estos servicios incluyen tiempo de procesador, sistema de archivos, comunicaciones, manejo de memoria, etc. En principio, el código fuente del kernel puede estar separado en distintas librerias y archivos pero una vez compilados estos archivos todo el código binario resultante, conocido como imagen, se carga en una zona de memoria específica. Todos los servicios se ejecutan en kernel mode.

2. Microkernel - El Microkernel, desarrollado como sucesor del Monolithic Kernel, se caracteriza por separar la mayor parte de los servicios en un conjunto de modulos independientes. El modulo principal contiene elementos básicos como el manejo de memoria, administración de de tiempo de procesador o scheduling y un sistema de mensajes o IPC básico para la comunicación entre módulos. Estos servicios se ejecutan en kernel mode. El resto de los servicios se incorporan de manera separada al momento del arranque y se ejecutan en user mode. Algunos de estos servicios son el sistema de archivos y un IPC mas robusto.

3. Nanokernel - El nanokernel no es un término completamente estandarizado. En principio, el enfoque del nanokernel es otorgar los servicios más necesarios usando solo unas cuantas lineas de código en comparación con las otras arquitecturas. Ade-

más, la arquitectura esta enfocada en la optimización de la velocidad del sistema. Una aplicación particular de este kernel es en los equipos de tiempo real. Debido a que el sistema operativo tiene que responder en tiempos perfectamente establecidos con la mayor precisión posible, este tipo de kernel es uno de los más adecuados.

4. Exokernel - El Exokernel es un diseño orientado hacia el desarrollo de aplicaciones y librerías que entran en contacto de manera mas directa con el hardware. En el caso de los kernels anteriores, los dispositivos de hardware son presentados al usuario como capas de abstracción, como es el caso del sistema de archivos o librerias gráficas como DirectX u OpenGL. El exokernel intenta mantener la interacción con el hardware lo mas directa posible sin perder las capacidades de administración como es el caso de MS-DOs que permitía ejecutar instrucciónes directamente al hardware pero no tenía servicio de multiusuario o multitarea. Este kernel esta siendo desarrollado por universidades como el MIT, su precursora, o la universidad de Cambridge. Actualmente ningún sistema operativo comercial tiene este tipo de kernel.

En la figura 3 al final del capítulo se hace una comparativa entre los diferentes tipos de kernels. Note que no se muestra el nanokernel, esto es debido a que no se a definido una estructura básica para este tipo de kernel.

1.4 Operating Systems

En este curso se estará hablando de los tres sistemas comerciales mas populares en las PC: Windows, Mac OS X y Linux. Además también se estudiará a sus tres versiones móbiles: Windows Phone, Android e iOS. Sin embargo, en cuanto a los aspectos internos solo hablaremos de Windows y Unix. Esta sección intentara aclarar el porque.

Conocer la arquitectura básica del sistema operativo con que se trabaja permite un mayor entendimiento de los resultados en rendimiento, velocidad y posibles errores en los desarrollos de apliciónes, asi como también permite desarrollar software mas fuertemente acoplado al sistema operativo como es el caso de los drivers. En primer lugar, hay que notar que los sistemas operativos para dispositivos móviles actuales no solo decienden de las versiones para PC sin que son estas mismas con solo algunas modificaciones. Por esto, se revisarán las diferentes arquitecturas que cada sistema operativo propone emparejando al mismo tiempo la versión PC y la versión móvil.

1.4.1. Windows

Los sistemas operativos windows han pasado por diferentes etapas de desarrollo, empezando por su version que operaba encima de MS-DOS desde Windows 1 hasta Windows 3.11. Después, usando la misma arquitectura se fucionaron los dos conceptos dando como resultado las versiones Windows 95, 98 y Me. Estas versiones estaban orientadas a usuarios domésticos. De manera paralela, Microsoft desarrollo un sistema para servidores con un kernel que sigue los objetivos de Unix como multi usuario y multi tarea. Este es el kernel de las versiones NT de Windows. Windows NT es un hybrid kernel que combina conceptos del monolithic y el micro kernel con funciones como multitasking. Las diferentes versiones de Windows para usuarios domésticos soportaban funciones parecidas pero el kernel no fue desarrollado en un principio con estas características, lo que daba como resultado un rendimiento mas pobre y algunos posibles errores. A partir de las version XP, se decidió por fusionar los desarrollos, dejando al kernel NT como la base de todas las versiones posteriores y limitando la compatibilidad de aplicaciones para versiones previas del sistema operativo. A partir de Windows XP y Windows 2000, la diferencia entre versiones radica en las apli-

caciones incluídas mientras que la arquitectura base es la misma.

La arquitectura base del kernel NT se muestra en la figura 4.

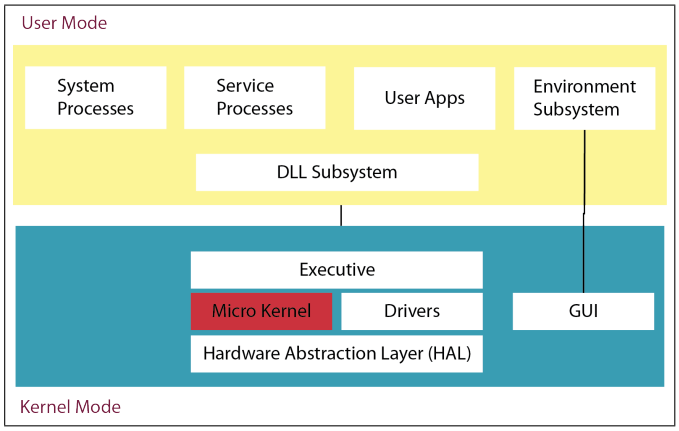


Figura 4: Diagrama del kernel Windows NT

Para sus versiones móviles, Windows tuvo tambié un desarrollo paralelo. Esto debido en parte a que la arquitectura de lo dipositivos móviles era diferente y no había un estandar claro. Los nombres de estas versiones fueron Windows Mobile Phone y Windows CE. Con el surgimiento de ARM como estandar para estos dispositivos, Microsoft decidió consolidar desarrollos al fusionar nuevamente sus dos productos. El kernel resultante se implementó en Windows 7 y Windows 8 así como la aparición de su nuevo sistema operativo Windows Phone 8. La ventaja del nuevo kernel es que puede ser compilado tanto para ARM como para x86. La arquitectura es muy similar, dependiendo en mayor medida de su plataforma .NET como API para los desarrolladores de aplicaciones.

1.4.2. Linux y Android

El sistema operativo Unix nace en la academia como una solución para la operación de main frames y terminales tontas que manejaban ditintas universidades. Este desarrollo fue de iniciativa académica que mas tarde sería imitado por empresas como AT&T. Hasta 19, el sistema operativo Unix fue desarrollado tanto por la industria como por la academia orientándolo a servidores y main frames. Esto dió lugar al nacimiento de diferentes versiones de Unix como AIX y Novell. Aunque este sistema operativo podía compilarse para arquitecturas x86, carecía de soporte para los diferentes dispositivos de hardware. En 19, Linus Torvals comienza como proyecto de maestría el sistema Linux basándose en la teoría desarrollada por Andrew Tannenbaum así como en el kernel del sistema operativo que el mismo Tannenbaum estaba desarrollando: Minix. En un golpe de suerte, Linus Torvals da a conocer a través de interntet su desarrollo, lo que atrae a Richard Stallman, fundador de la fundación para software libre GNU. Stallman estaba buscando crear un sistema operativo que pudiera liberar su código fuente a todo aquel interesado y así liberar la creatividad de los desarrolladores de licencias restrictivas. Gracias a desarrolladores en todo el mundo y el apoyo de GNU y la despues formada Free Software Foundation nace GNU/Linux o Linux. El kernel de Linux sigue el estandar de Unix, siendo un 100 % compatible con el código fuente desarrollado para Unix, salvo excepciones. Linux es un Monolithic Kernel a diferencia de Minix que es un Microkernel. La structura básica se muestra en la figura 5.

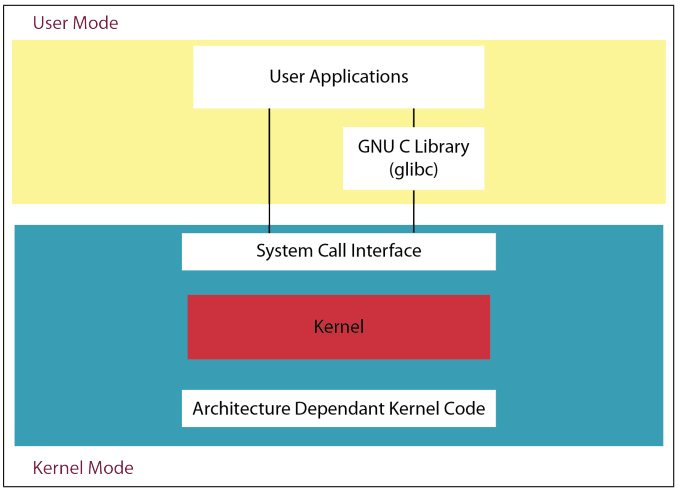


Figura 5: Diagrama del kernel de Linux

En principio, Linux incluía el soporte a dispo-

sitivos o drivers dentro de su kernel. Esto obligaba a recompilar el kernel de un sistema ya implementado si se requería de agregar un dispositivo nuevo. En la actualidad se le ha dotado con un sistema más modular que permite agregar nuevos dispositivos sin tener que recompilar. Aún así el kernel sigue siendo tipo monolithic.

Android es un desarrollo hermano o Fork del kernel de Linux por parte de Google. Ya que una de las ventajas de Linux es que su código es en su mayoría portable a otras arquitecturas, ha sido la elección de muchas empresas para crear sistemas operativos móviles. En la actualidad Android de Google ha sido el que a prevalecido. Su arquitectura básica se muestra en la figura 6.

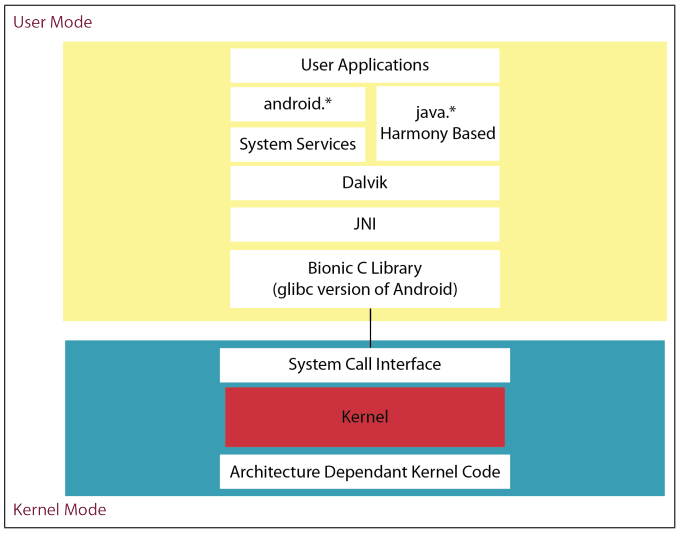


Figura 6: Diagrama del kernel de Android

Como puede observarse. Android utiliza como base el kernel de linux y usa un stack de la tecnología java de oracle como API e interfaz para los desarrolladores. La maquina virtual java (JVM) funciona como un segundo kernel que funciona encima de otro kernel. La JVM usada por Google esta desarrollada especialmente para Android y se llama Dalvik. La primera capa JNI es usada para hacer interfaz entre la máquina virtual y las aplicaciones nativas en linux. Cabe mencionar que la máquina virtual o JVM esta desarrollada por Google y tiene pornombre . Esta JVM no es la misma quelp oa desarrollada por Oracle. Además de la libreria estandar de Java, Jvm contiene una librería específica para dispositivos android. Esa solución es parecida a la implementada en Windows Phone 7, sin embargo windows phone permite desarrollar aplicaciones nativas sin jacer uso extensivo de .NET mientras que en Android esto no es el objetivo.

1.4.3. Mac OS X y iOS

El desarrollo de Mac OS X tiene lugar en un momento clave de la compañía Apple. La estructura se muestra en la figura 7.

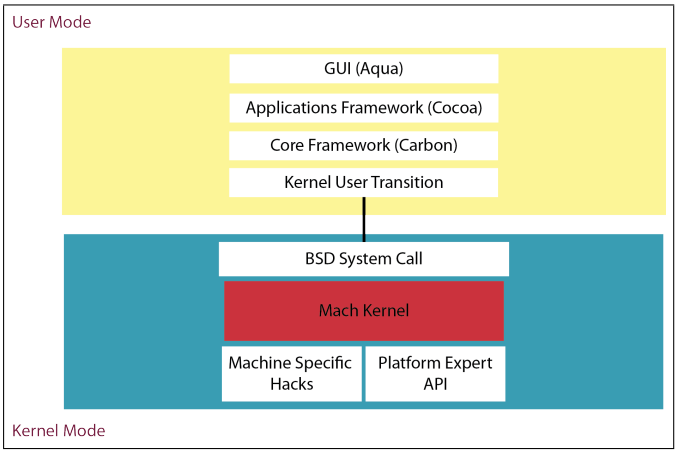


Figura 7: Diagrama del kernel de Mac OS X

Hasta 1999, Apple había vendido computadoras con una arquitectura diferente a la x86 desarrollada por IBM llamada PowerPC. Esto puede paracer confuso ya que la PC fue desarrollada por IBM también, pero IBM solo licenció la arquitectura x86. Actualmente las patentes x86 son otorgadas a Intel y AMD solamente a diferencia de ARM que es open source. Regresando al tema de Apple, en 19 se enfrentó a una crisis ante omniprecencia de la PC. Su antiguo CEO y fundador Steve Jobs se encontraba separado de Apple y manejando la empresa NeXT. En esta empresa se desarrollo un nuevo Unix usando la tecnología de microkernel basándose en otro sostema operativo desarrolado en la academia llamado Mach. Esta tecnología era novedosa y sin

implementar en sistemas comerciales hasta el momento. El sistema operativo tuvo por nombre NeXTStep. En una última jugada, Apple adquiere NeXT junto con todas sus patentes para hacer frente a la crisis, regresando a Steve Jobs como CEO. El previo sistema operativo de Apple, Mac OS, es absorbido por NeXTStep y usando los conceptos de Mach crean el kernel llamado Darwin, base del sistema operativo Mac OS X. Como puede verse en la figura 7, Mac OS X es la fusión de conceptos de Mach con otro sistema Unix llamado BSD. Aunque esto se ve en principio como una solución pobre, el desarrollador no tiene que lidiar con esto ya que solo necesita hacer uso del API escrito en Objetive C y Cocoa. Objetive C es un subconjunto de C similar a C++ pero con una mayor orientación hacia el uso de objetos. Cocoa es la interfaz o Framework que usa Objetive C para comunicarse con el kernel darwin. Objetive C tambien fue desarrollado por NeXT antes de la compra de Apple.

La versión móvil de Mac OS X es iOS. iOS utiliza la misma arquitectura que Mac OS X. la única diferencia son las implementaciones de los servicios. Un ejemplo es el sistema de archivos de iOS que esta orientado a memorias flash solamente mientras que Mac OS X esta orientado a múltiples medios de almacenamientos. Algo más evidente es el uso de diferentes interfaces de usuario y sus APIs localizados en la capa superior. Mientras que Mac OS X utiliza una libreria GUI llamada Aqua, iOS utiliza SpringBoard.

Como se puede notar, solo dos estándares están presentes en los diferentes kernels: el propuesto por Microsoft y el estandar Unix. Por este motivo la discusión en los siguientes capítulos se centrará en estos dos estándares y en la JVM, la cual esta presente en todos los sistemas operativos y se puede considerar como una implemetación mas de un sistema operativo. Sin embargo, se mencionarán las diferencias pertinetes en los seis sistemas operativos de ser necesario.

1.5 Bootloader

El proceso de arranque de la computadora consiste en la inicializacion de los dispositivos y la carga del sistema operativo en memoria. Este proceso, aunque estandar, no fué igual en los inicios de las computadoras.

En la época previa a las computadoras personales y a los dispositivos de entrada actuales, el proceso de arranque variaba según las necesitades del hardware y en su mayoría no contaban con un sistema operativo en disco. El proceso de arranque que usaban iba de no contar con un sistema operativo y el acceso a hardware era directo a memoria a traves de palancas y botones hasta pequeñas secuencias de instrucciones programadas dentro de circuitos integrados o sistemas basados en el uso de tarjetas perforadas. En la década de 1970, diferentes compañias se dieron a la tarea de crear sistemas operativos que ofrecieran las herramientas para explotar el hardware en el que fueran instalados. En esta época era común ver computadoras funcionando usando interpretes de lenguaje como BASIC o LISP donde el operador tenía que conocer tanto el lenguaje como la arquitectura con la que estaba trabajando.

A mediados de la decada de los 70, IBM comenzo a implementar un plan comercial conocido como ``chess project'' que más tarde sería el lanzamiento de la primera computadora personal. Dentro de este plan se contemplaba que para poder capturar el mercado se debería usar una arquitectura abierta y flexible además de tener la facilidad de usar la computadora sin conocimientos profundos de su funcionamiento. IBM se acerco a Bill Gates, entonces CEO de Microsoft, con la petición de crear un sistema operativo que pudiera ser cargado desde discos magnéticos, producto que IBM acababa de lanzar al mercado. Este sistema operativo debería tener la capacidad de trabajar con diferente tipo de hardware y poder presentar al usuario diferentes aplicaciones sin la necesidad de conocer un lenguaje de programación. A su vez, la crea-

ción de un sistema operativo portable abriría paso a un desarrollo más ágil de hardware. Microsoft compra de la compañia Digital Research un prototipo de un sistema operativo para discos, DOS por sus siglas en inglés. El sistema inicial, denominado QDOS o Quick Dirty Operating System fué modificado y preparado para su distribución comercial. Dos sistemas fueron lanzados tanto por IBM, el PC-DOS, como por microsoft, el MS-DOS. Más adelante otras compañias desarrollarían sus propios sistemas, entre ellos cabe mencionar Apple y Novell. En respecto al sistema Unix, este fué concebido previo a la IBM-PC pero luego fué adaptado para ser compatible con este tipo de hardware.

Este tipo de sistemas, que son el defacto estandar de la industria para computadoras personales o PCs compatibles con IBM, funcionan en conjunto con el hardware de la computadora para poder inicializar el sistema. Para poder darle la habilidad al hardware de arrancar con diferentes tipos de sistemas operativos, el arranque se dividió en tres pasos: inicialización del hardware, carga del sistema operativo e inicialización del sistema operativo. Existen otros acercamientos como el caso del sistema operativo Linux que en el tercer paso reinicializa el hardware. El primer y segundo paso son denominados bootstrap. La inicialización del hardware es llevada acabo por el Basic Input-Output System o BIOS que es almacenado unidades ROM o Flash empotradas en el hardware. Aunque el nuevo estandar, Extensible Firmware Interface o EFI, esta reemplazando el antiguo BIOS, llamado ahora legacy BIOS, se estará usando el término BIOS para ambos sistemas a menos que se mencione lo contrario en el resto del curso.

El BIOS se encarga también de cargar en memoria el programa de carga o bootloader de una dirección particular de las unidades de disco. En la actualidad el BIOS puede cargar el bootloader dese múltiples sistemas de almacenamiento. El bootloader se encarga de buscar y cargar el kernel del sistema operativo en memoria. Para poder escribir un bootloader se deben considerar ciertas condiciones previas impuestas por el BIOS.

Primero, el BIOS busca el bootloader en el primer sector de la primera pista de la primera cara del disco. Segundo, el BIOS requiere de una firma especial al final del sector, los últimos dos bytes deben contener el valor `0xAA` y `0x55`[1] respectivamente. Esta condición también introduce una restricción de tamaño, el bootloader debe tener un tamaño de 512 bytes o un sector, contando los dos últimos bytes de la firma. Una restricción también impuesta a nivel de diseño es la retrocompatibilidad del BIOS con los sistemas operativos de 16 bits, lo que restringe al cargador a operar en modo de 16 bits y limitando el uso de memoria a 2^16 direcciones de memoria, es decir, del `0x0000` al `0xFFFF`. Una última característica de diseño es que el BIOS cargará el bootloader en la dirección `0x7C00`. Por lo tanto, todas las direcciones invocadas por el bootloader tienen como referencia la dirección de carga. La razón de esta dirección en particular es debido a que todos los dispositivos de I/O deben de ser asignados a una dirección de memoria para su posterior invocación además de dejar espacio de sobra para cargar el kernel. Esta desición fué tomada por los ingenieros de IBM al desarrollar la IBM PC 5150, dejando el último kilobyte (1KB) de la memoria para el bootloader. En ese entonces el mínimo para correr el PC-DOS era 32KB[2] .

En la tabla se muestra un bootloader que imprime la letra ``A''.

Código 1: Un simple bootloader

```
1 [BITS 16]
2 [ORG 0x7C00]
3
4 MOV AL, 65
5 CALL PrintCharacter
6 JMP $
7
8 PrintCharacter:
9 MOV AH, 0x0E
```

[1] La notación `0x` hace referencia a un número hexadecimal

[2] `0x7C00` es el inicio del último KB en 32KB también es representado como (`32KB - 1KB`)

```
10 MOV BH, 0x00
11 MOV BL, 0x07
12
13 INT 0x10
14 RET
15
16 TIMES 510 - ($ - $$) db 0
17 DW 0xAA55
```

Las primeras dos líneas de código son directivas de compilador que indican la operación en modo de 16 bits y que las direcciones usadas son relativas a la posición inicial dada por el BIOS.

Ahora se presenta el problema del despliegue en pantalla. Debido a que no tenemos el kernel cargado, no podemos hacer uso de ninguna librería. Para poder imprimir en pantalla haremos uso de subrutinas de hardware llamadas interrupciones. Una interrupción de hardware es invocada por la instruccion `INT` seguida del número de la interrupción. La impresión en pantalla toma cuatro argumentos: el caracter a imprimir en el registro `AL`, el tipo de servicio en `BL`, colores del caracter en `BH` y. De todas las propiedades tal vez la más confusa es el tipo de servicio almacenado en `BL`. La interrupcion `0x10` que es la que vamos a invocar proporciona múltiples servicios para la impresión en pantalla. El tipo de servicio es indicado por un número de servicio definido en la arquitectura del BIOS. Las siguientes dos lineas del código invocan a una subrutina preparada para la impresión de un caracter en pantalla:

```
CALL PrintCharacter
JMP $
```

La primera linea almacena en `AL` el caracter a imprimir en código ASCIII, 67 para la letra A. La segunda línea llama a la subrutina de impresión.

```
MOV BH, 0x00
MOV BL, 0x07

INT 0x10
RET

TIMES 510 - ($ - $$) db 0
DW 0xAA55
```

El cuerpo de la subrutina prepara los datos para llamar a la interrupción necesaria. `AH` almacena el tipo de servicio para impresión en pantalla `0xE0`. `BL` contiene la página en la que se encuentra la zona de memoria del monitor. Por el momento este concepto se dejará para temas más adelante, no se preocupe en entenderlo. El último dato que describe el apecto del caracter a imprimir se coloca en `BH`. Se eligió color gris sobre fondo negro. Por último se llama a la interrupcion `0x10`[3].

Por último se cicla a la computadora en un ciclo infinito con la instrucción:

Las siguientes instrucciones requiere un poco mas de explicación:

Note que nuestro programa debería de hacer un ciclo infinito en la última instrucción descrita. Por lo tanto estas dos lineas no se alcanzarían en un flujo de ejecución. Sin embargo estas dos instrucciónes son directivas de compilador. La primera nos sirve para rellenar nuestro código binario resultante para que ocupe 510 Bytes. El símbolo `$$` apunta a la dirección inicial de nuestro programa. Por lo tanto calculamos el espacio total que ocupa nuestras instrucciones contando la actual y el total se lo restamos a 510. Esto nos da la cantidad necesaria de `0` colocados en el código máquina al ser compilado. La siguiente instrucción coloca en el código máquina resultante la firma del bootloader en los dos últimos bytes.

Para crar el código máquina necesitamos un compilador de ensamblador, en windows, linux y osx podemos usar NASM[4]. Para compilarlo en NASM se utilíza la siguiente instrucción

Código 2: Compilando el bootloader con NASM

```
nasm aboot.asm -f bin -o boot.bin
```

con esta instrucción le pedimos la compilador una salida binaria sin encabezado de nuestro programa. El tema de encabezados se vera

[3] en lenguaje ensamblador los numeros hexadecimales se denotan por n dígitos seguidos de la letra H

[4] `http://www.nasm.us`

más adelante. El último paso es colocar el programa obtenido en el MBR. Unix tiene por defecto la instrucción de consola `dd`. En esta ocasión lo colocaremos en una unidad virtual para ser cargada con Virtual Box [5]. La lista de comandos siguientes en linux sirven para crear una imagen virtual de un floppy, crear un directorio donde montar el floppy, montar el floppy, escribir nuestro bootloader y desmontar la unidad [6].

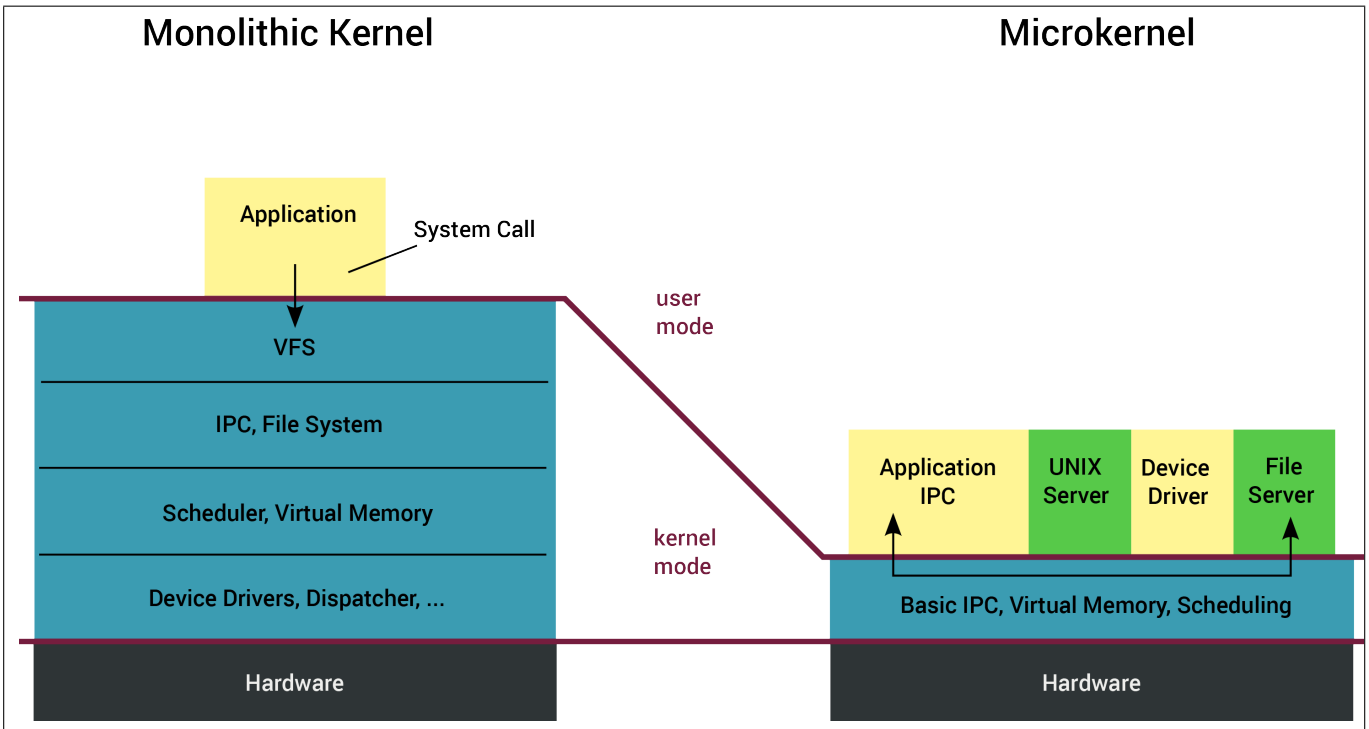
Código 3: Preparando un floppy virtual

```
mkfs.dos -C floppy.img 1440
sudo mkdir /mnt/vfloppy
sudo mount -o loop floppy.img /mnt/vfloppy
sudo dd if=boot.bin bs=512 of=/mnt/vfloppy
```

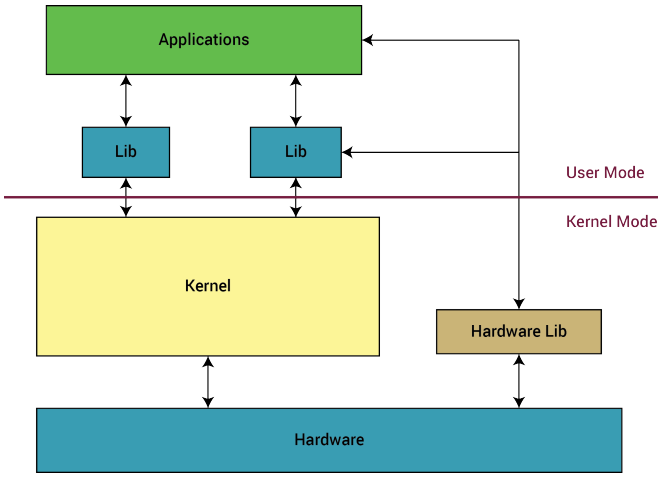
Por último colocamos el floppy en la máquina virtual e iniciamos la máquina virtual. Despues del self test de la maquina virtual podemos ver como la A aparece en pantalla y la máquina virtual queda suspendida en un ciclo infinito.

[5] `http://www.virtualbox.org`

[6] Las instrucciones se tomaron en cuenta en una instalación de linux sobre una máquina virtual como se muestra en los videos del curso con liga



(a) Comparativa entre el monolithic y el micro kernel



(b) Estructura del Exokernel

Figura 3: Comparativa De Los Diferentes Tipos de kernel

2 Memory Management

Una de las actividades principales del kernel es el manejo de memoria RAM. En lo subsecuente se hará referencia a la memoria RAM solamente como memoria del sistema. Como puede deducirse después de estudiar el arranque de una PC, el sistema operativo debe de proveer una capa de abstracción para el acceso a memoria. La opción mas sencilla es dejar que cada aplicación tenga acceso directo a la memoria y la maneje de manera independiente al resto de las aplicaciones. Sin embargo esto nos enfrenta a diferentes problemáticas. En principio la memoria tendría que manejarse a bajo nivel usando direcciones físicas. El manejo de direcciones físicas dificultaría el desarrollo de aplicaciones. Además abriría la posibilidad de que una aplicación tenga acceso al espacio de memoria de otra aplicación y sobreescribirla o peor aun, sobreescribir el mismo kernel o los servicios del BIOS. Por esta razón se opta en el diseño de kernels por crear una capa de abstracción de la memoria apuntando a incluir las siguientes características:

1. Seguridad
2. Transparencia
3. Eficiencia en el manejo del espacio disponible
4. Manejo de grandes volúmenes de aplicaciones activas
5. Flexibilidad

Estos objetivos intentan dotar al sistema operativo con una capa de abstracción de manejo de memoria que le permita ofrecer un ambiente confiable y eficiente a cada aplicación además de ser lo suficientemente flexible para ejecutar aplicaciones que requieran más memoria de la que físicamente se encuentra disponible. Para poder explicar como se resuelven todos estos puntos, primero se describirá como se encuentra direccionada la memoria física.

2.1 Memory Addressing

La arquitectura actual de procesadores usa un modelos de memoria física conocido como modelo Von Neuman. En este modelo se tiene un solo espacio de memoria que es compartido tanto por el código ejecutable como por los daros. Existen diferentes métodos para manejar la memoria física que son soportados de manera nativa para contribuir con la creación de una capa de abstración para el manejo de memoria por parte del sistema operativo. El controlador de memoria o Memory Management Unit (MMU)de la arquitectura x86 soporta dos modos de operación: Real Mode y Protected Mode. Real Mode es un modo de operación con acceso directo a las direcciones físicas, este modo se incluye para mantener una compatiblidad hacia atrás con los antiguos sistemas operativos y para el arranque usando un bootloader. El modo protegido, que es en el que estaremos trabajando, se describe a continuación. En Protected Mode se utilizan tres tipos de direcciones:

1. Logical Address. Estas direcciones son las manejadas a nivel de lenguajes de bajo nivel como Assembler y se forman de un segment y un offset. Un segment es la direccion inicial del bloque de memoria asignado a un proceso[7] y un offset es el desplazamiento a partir de la dirección inicial contenida en segment. Usualmente solo se debe tener conciencia de estas direcciones cuando se manejan lenguajes o librerías de bajo o medio nivel como C. Estas son las únicas direcciones a las que tenemos acceso en Protected Mode.

2. Linear Address. Este tipo de direcciones etiquetan las zonas de memoria usando un único índice de 32bits o 64bits dependiendo el tipo de arquitectura. 32 bits pueden diccionar hasta 4GB de memoria mientras que

[7] A partir de ahora se hará referencia a procesos en lugar de aplicaciones, en el siguiente capítulo se aclarará sistemáticamente la diferencia entre ambos.

64 bits pueden direccionar 2B aproximadamente. Esta es la razón por la que sistemas operativos de 32 bytes tienen que usar una modificación especial cuando requieren direccionar memoria mas allá de los 4GB.

3. Physical Address. Esta es la verdadera dirección de una celda de memoria. Esta dirección hace referencia a la señal eléctrica de los pines de la memoria y se representa con un `unsigned int`. De 32 bits o 36 bits para procesadores de 32 bits y 64 bits o para procesadores de 64 bits.

El acceso a memoria a nivel de hardware es manejado por el Memory Arbiter. Esta pieza de hardware permite a cada CPU presente acceder a la memoria así como también a los diferentes dispositivos usando Direct Memory Access sin necesidad de solicitarlo al CPU. Cada CPU requiere un Memory Arbiter dedicado. La physical address es obtenida a nivel de hardware

2.2 Segmentation

Segmentation fue el primer método introducido para incorporar mecanismos de memoria virtual y seguridad en el manejo de memoria. Este mecanismo usa direcciones llamadas Logical Address. En la arquitectura Intel, segmentation es soportado desde el modelo 80286. Para poder entender la manera de calcularlo es necesario revisar la estructura binaria de las Logical Address. El primer paso es obtener la linear address utilizando la logical address. El hardware de la arquitectura x86 calcula de diferente manera la linear address en sus diferentes modos de operación. Para protected mode, el hardware utiliza una estructura de 16 bits para el segment o segment selector y 32 bits para el offset en procesadores de 32 bits. Mientras que el offset es un unsigned int que informa el desplazamiento desde la dirección inicial asignada, el campo de segment tiene una estructura más compleja. La estructura se muestra en la figura .

Los bits están enumerados de derecha a izquierda a modo little indian como es el estandar de la arquitectura x86, siendo el bit menos significativo o LSB por las sigla en inglés de Less Significant Bit el bit de más a la derecha con índice `0` y el bit más significativo o MSB por las siglas en ingles de Most Significant Bit con índice 15 en el caso del segment selector. Los 13 bits mas significativos son llamados index y mantiene el índice de una entrada slmacenada en una Table Descriptor explicada más adelante. El bit 2 indica el tipo de Table Descriptor a la que apunta y por úiltimo los bits 0 y 1 indican el nivel de permiso que tiene el segmento. linux en particular usa solo el valor 0 para kernel mode y 3 para user mode. A cada proceso creado se le otorgan tres segmentos: el code segment donde reside el código ejecutable, el stack segment donde se almacenan valores dinamicamente a manera de pila y el data segment donde se almacenan valores a manera de acceso indisado. La diferencia entre el data segment y el stack segment reside en la manera de acceder a los datos. El procesador mantiene los valores de los distintos segment en los llamados registros de segmento. Son seis registros de segmento, tres de uso particular y tres de uso general. Los tres registros de uso particular son: `ds` para el data segment, `ss` para el stack segment y `ds` para el data segment. Los tres registros de propósito general son: `es`, `fs` y `gs`. Para poder traducir rapidamente de la logical address a la linear address, el sistema operativo utiliza un campo de 8 bytes llamado segment descriptor. Los segment descriptor son almacenados en tablas. Las tablas que almacenan los segment descriptor pueden ser de tipo Global Descriptor Table (GDT) o de tipo Local Descriptor Table (LDT). Usualmente solo se mantiene una GDT por cada procesador presente y su tamaño y dirección son almacenados en el registro gdtr. Las LDT son creadas por los procesos que requieren de reservar mas memoria y su tamaño y dirección son almacenados en el registro `ldtr`. Cada proceso tiene una sola tabla LDT. Existen varios tipos de segment descriptor pero el hardware mantiene solo tres estructuras físicas. La estructura física para los procesado-

res de 32 bits se describe en la figura .

La tabla 1 describe los diferentes campos de los segment descriptors.

Como se ha mencionado, existen diferentes tipos de segment descriptor. Aunque físicamente se usa la misma estructura, las banderas indican el tipo de descriptor que se esta almacenando. Este tipo de prácticas solo es válido cuando se trabaja a este nivel de abstracción. En lenguajes de mas alto nivel se deberían crear estructuras de datos con diferentes nombres. La manera de manejar los segment descriptor quedan a discreción del sistema operativo. En eñ caso de linux se manejan cuatro tipos:

- Code Segment Descriptor. Este describe un code segment. La bandera S esta encendida y la bandera Type almacena el valor que representa un code segment descriptor. Las otras banderas dependen del proceso al que pertenece. El segment puede encotrarse en una GDT o una LDT.

- Data Segment Descriptor. Este describe un stack o data segment. La bandera S esta encendida y la bandera Type almacena el valor que representa un data segment descriptor. Para linux, el data y el stack segment no necesitan diferenciarse para calcular su linear address, debido a esto solo se usa un tipo de segment descriptor. El segment puede encontrarse en una GDT o una LDT.

- Task Segment Descriptor. Describe el estado de un proceso. En linux un proceso se suele llamar Task o Thread. Este solo se puede encontrar en la GDT. El campo Type contiene 11 o 9 dependiendo de si el proceso se encuentrs en el CPU o no respectivamente. La bandera S tiene el valor de 0.

- Local Table Table Descriptor. Este descriptor indica que el segmento apunta a una LDT. La bandera S esta apagada y Type tiene el valor de 0. Solo se puede almacenar este tipo en una GDT.

El procesador x86 mantiene registros no programables por cada segment register que almacena los correspondientes segment descriptors. El CPU puede usar estos registros no programables para traducir las logical address en linear address. Estos registros no programables se escriben automáticamente al momento de que un proceso entra en ejecución. Cuando el CPU se ve en la necesidad de calcular la linear address de una logical address que no esta en ejecución utiliza el algoritmo mostrado en la tabla .

Los pasos del algoritmo son los siguientes:

1. El CPU revisa el campo TL del segment selector para determinar si el descriptor esta en la GDT o en la LDT. La dirección de la tabla la obtiene de los registros gdtr o ldtr. Note que el registro ldtr se tiene que reescribir cada que un proceso entra en ejecución. El gdtr se mantiene con el mismo valor desde que el kernel se carga en memoria.

2. El índice del segment descriptor se obtiene multiplicando el valor almacenado en la bandera index del segment selector por 8 que es el tamaño en bytes del segment descriptor y sumándole la dirección de la tabla obtenida en el paso anterior.

3. por último, la linear address se obtiene de sumar el offset de la logical address con la linear address almacenada en el segment descriptor.

En el caso de que la logical address pertenezca a un proceso en ejecución se omiten el primer y segundo paso al tomar el segment descriptor de los registros no programables. La tabla muestra un diagrama de la memoria indicando a donde apuntan las distintas direcciones. Estas operaciones sirven debido a que asumimos que el direccionamiento de memoria es linear y empieza en 0.

En general, los sistemas operativos actuales no hacen uso exteso de este mecanismo debido a que otras arquitecturas diferentes a la x86 no hacen uso de la segmentación. Esta decisión

Nombre	Descripción
Base	Contiene la linear address del primer byte del segment
G	Granularity Flag. Si esta en 0 el tamaño se expresa en bytes. Si esta en 1 el tamaño se expresa en múltiplos de 4096 bytes
Limit	Contiene el desplazamiento al último byte del segment indicando asi también el tamaño del segmento. Si G tiene el valor de 0 el tamaño va de 1 byte hasta 1 MB. Si G es 1 el tamaño va de 4 KB hasta 4 GB.
S	System Flag. Si es 0, el segment descriptor es un system segment descriptor. Si es 1 es un data o code segment descriptor
Type	El tipo de segment descriptor y sus permisos
DPL	Tipo de privilegio necesario para acceder a este segment descriptor. Si es 0 solo puede accedre el kernel (kernel mode). Si es 3 cualquier proceso puede acceder al segment descriptor (user mode)
P	Present Memory Flag. Indica si: 0, el segmento no está en la memoria o 1 si se encuentra. Este concepto esta relacionado con el swapping entre la memoria y los dispositivos de almacenamiento. El swapping no necesariamente ocurre con segments completos, por lo que sistemas como Linux siempre ponen el valor de 1 ya que nunca hace swapping a todo el segment.
D or B	indica si el offset del segment es de 32 bits cuando esta en 1 o de 16 bits cuando esta en 0
AVL	Bandera de propósito general para el sistema operativo.

Cuadro 1: Flags del Segment Descriptor

simplifica el trabajo de transportar el sistema operativo a otros dispositivos con arquitecturas diferentes como es el caso de los dispositivos móviles con arquitectura ARM que no soporta segmentación.

2.3 Paging

Paging es un segundo método para crear un modelo virtual de memoria. Este permite que cada segment asignado a un programa no necesite estar en direcciones físicas contiguas como es el caso de usar segmentation. Para matener la eficiencia en el direccionamiento de memoria, el kernel agrupa múltiples linear address en una unidad llamada Page. Todas las linear address dentro de una page apuntan a direcciones físicas contiguas. Esto simplifica el uso de permisos ya que solo se necesita llevar un registro del tipo de permisos por Page en lugar de por linear address. Este mecanismo es llevado a cabo a nivel de hardware por la Paging Unit. La paging unit divide a la memoria RAM en bloques llamados Page frames. El tamaño del page frame es el mismo de una page, lo que permite al CPU colcar una page en cada page frame solamente. La diferencia entre una page y un page frame es que una page es un grupo de linear address y los datos almacenados en cada linear address y un page frame es un espacio de almacenamiento en memoria RAM. Una page puede ser almacenada en otro lugar que no sea la memoria RAM.

La arquitectura x86 utiliza dos métodos para implementar paginación. El primer método, o Paging, divide a la linear address en tres bloques. El primer bloque de los diez bits mas significativos, es decir el bit 31 al 22, es llamado Page Directory. Los siguientes diez bits son llamados Page Table y los últimos doce bits son llamados offset. La intención de usar dos niveles es de re-

ducir el uso de memoria por tabla. Este esquema permite mantener una tabla de 2^10 o 1 MB para almacenar los Table Directory y cada Table Index es cargada a memoria cuando es requerida por un proceso. Tanto la Table Directory como la Table Index tienen la misma estructura. Este método permite Pages de 4KB de tamaño[8]. Esta estructura es mostrada en la tabla 2.

Las funciones de las banderas se explican a continuación. Los nuevos conceptos que aparacen en la tabla se iran explicando a lo largo del resto del capítulo.

El concepto de swapping mencionado en las banderas Present, Dirty y Accessed es parte del me anismo de memoria virtual por el que fue concebido el Paging. Sin olvidar la seguridad. Swapping es el mecanismo que intercambia una page contenida dentro de un page frame a un dispositivo de almacenmiento secundario y pone ptra page en su lugar. Esto se utiliza para extender el espacio disponible en memoria en ciertos casos. El sistema de memoria virtual se concibio debido a que, desde la creación de las computadoras, la memoria RAM siempre ha sido mas cara que los dispositivos de almacenamiento magnético. El sistema de memoria virtual nos permite extender la memoria disponible para procesos que requieran una cantidad inferior a la memoria física existente. Este método no sirve si se requiere de un bloque contiguo que exceda el tamaño de la memoria física. La desventaja de este método es el que el acceso a dispositivos de almacenamiento es varias veces mas lento que a memoria. Esto provoca que un sistema que haga mucho swapping vea su rendimiento reducido ya que involucra al menos tres acciones: solicitud de una page, excepción de page fault por la paging unit, swapping por el sistema operativo. El sistema operativo se encarga de decidir que page intercambia con una en el dispositivo de almacenamiento al ver las banderas Dirty y Accessed. El sistema operativo usa la page es la menos usada como candidata a swapping.

El segundo método de paging se conoce como extended paging. Este método se incorporó en la arquitectura x86 a partir del modelo Pentium. En extended paging se usan pages de 4 MB en lugar de 4 KB. Esto se logra dividiendo la logical address en dos niveles en lugar de tres. Los primeros 10 bits mas significativos son el Page Directory y los siguientes 22 son el offset de la physical address. Los bits de los campos del Page Directory siguen siendo los mismos excepto que el campo de la physical address solo se toman los primeros 10 bits mas significativos debido a que estamos manejando múltiplos de 4 MB y los 22 bits menos significativos de estos múltiplos siempre se encuentran apagados. Cual método usar depende de la aplicación. El método de Paging decrece en rendimiento cuando un proceso solicita bloques de memoria grandes. Esto es si el proceso accede constantemente a las diferentes pages que componen el bloque de memoria que solicitó, lo que puede provocar múltiples excepciones de page fault. Extended Paging presenta un mejor rendimiento en estos casos al reducir el número de excepciones para este tipo de procesos. En otros casos Paging presenta un mejor rendimiento promedio. Extended paging tambien presenta una ventaja en el uso de cache explicado mas adelante.

Un problema que se presentó previo a la aparición de procesadores de 64 bits es cuando se requeria direccionar mas de 4 GB de memoria física ya que solo se manejaba una dirección de 32 bits[9]. Este problema aparece primero en servidores y mas adelante en equipos domésticos con el abaratamiento de la memoria RAM. Para solucionar esto, Intel introduce el mecanismo llamado Physical Address Extension (PAE) Paging a partir de los procesadores Pentium Pro. Para esto, aumenta el número de pines de la memoria de 32 a 36 e introduce un nuevo nivel de paging creando una tabla especial con entradas

[8] el offset usa 12 bits, $2^12 = 4096$

[9] $2^{32} \simeq 4$ GB

Nombre	Descripción
Present	Tiene el valors 1 si la Page se encuentra en memoria. Si es 0 el resto de los campos no tienes significado. En caso que un proceso solicite esta page la paging unit levanta una excepción llamada page fault.
Physical Address	Los siguientes 20 bits representan las physical address en una Page Table. Solo se necesitan 20 bits en lugar de 32 porque las pages son de 4 KB de tamaño. Por lo tanto la dirección inicial de una page es un múltiplo de 4096 y los 12 bits menos significativos estarán apagados. En una Page Directory apunta a la Page Table que contiene la physical address.
Accessed	Toma el valor de 1 cuando la paging unit la usa para calcular una dirección solicitada. Este campo es usando junto con Present para implementar swapping.
Dirty	Toma el valor de 1 cuando alguna localidad de una Page Table es escrita. También se usa para swapping
Read/Write	Permisos de la page table, puede ser Read o Read/Write
User/Supervisor	Indica el nivel de permiso requerido para pode acceder a la page.
PCD y PWT	Estas dos banderas indican como debe ser administrada la page cuando esta en memoria cache.
Page size	Indica el método de paginación
TLB	Esta bandera también se usa para el manejo de cache.

Cuadro 2: Flags del Table Directory

de 64 bits que puedan almacenar los 36 bits necesarios. Note que se usan 64 bits en lugar de los 36 ya que los procesadores solo manejan localidades con tamaño múltiplo del tamaño el bus, es decir 32 bits en este caso.

Para procesadores de 64 bits se mantuvo el tamaño de las pages en 4KB excepto en ciertas arquitecturas como la alpha de o la ia64 de Intel para servidores donde el tamaño es variable. Para mantener la eficiencia en estas arquitecturas se decidió introducir mas niveles de paging. Para la versión de 64 bits de la arquitectura x86 (x86_64) se manejan 4 niveles de paging, separando la linear address en cuatro bloques de pages de 9 bits cada uno y un offset de 12 bits (4 KB). La cantidad de niveles varia de arquitectura en arquitectura, un ejemplo de esto es la arquitectura PowerPC de IBM usada en las antiguas computadoras apple y la Playstation 3 que usan tres niveles de paginación separando la linear address en dos bloques de pages de 10 bits, un bloque de page de 9 bits y un offset de 12 bits. Note que ninguna de las arquitecturas mencionadas usan 64 bits para la linear address. Por el momento, se consideró que 46 bits en la arquitectura x86_64 pueden satisfacer la demanda en direccionamiento de memoria. Esto equivale a direccionar 2^{46} direcciones físicas lo que es al rededor de.

2.4 Caching

La velocidad de la memoria es hasta ahora menor que la de los procesadores. Esto implica que el procesador queda esperando a la memoria por la transferencia de una localidad. La diferencia de velocidades crea un cuello de botella. Para poder reducir este efecto, se creó la memoria cache. Este tipo de memoria es mas rápida debido a que es de tipo Static RAM en lu-

gar de la usada comunmente denominada Dynamic RAM. La cache se coloca dentro del dado el procesador dandole mayor ventaja al estar mas cerca del CPU. La implementación de este tipo de memoria usa el llamado Principio de Localidad. Este principio hace referencia a que despues de acceder a una localidad de memoria, es altamente probable que se enseguida se acceda a alguna localidad cercana. Por esto, la cache esta formada por varias unidades llamadas lines. Cada line puede almacenar varias localidades de memoria. La transferencia entre la memoria principal y la cache sucede en rafagas donde se transmiten varias localidades juntas. El modo de operación es similar al de Paging. La cache es admininstrada por el cache controller. El cache controller almacena un conjunto de bits llamado Tag para identificar las localidades de memoria de cada line. Cuando el procesador requiere una localidad, el cache controller revisa el tag asociado y lo compara con los tag de cada line. Si lo encuentra se tiene un cache hit, en caso contrario se tiene un cache miss. Cuando ocurre un cache hit existen dos comportamientos dependiendo de la operación a realizar. Cuando es de lectura, el cache controller transfiere la localidad al CPU sin acceder a la memoria RAM. En el caso de que la operación sea de escritura existen dos estrategias: write through y write back. En write through, el cache controller escribe la line tanto en la cache como en la memoria principal. En write back, el cache controller solo escribe la line en la cache, esperando hasta que exista una solicitud de flush por parte del CPU para escribir las lines modificadas a RAM. La solicitud flush ocurre principalmente en un evento de cache miss.

Cuando ocurre un cache miss, se realiza un flush de ser necesario y se sobrescribe una linea por la solicitada por el CPU. En equipos con multiples nucleos con cache propia y memoria RAM compartida, el cache controller siempre tiene que revisar en una operación de escritura las cache de los otros procesadores para actualizarla en caso de ser necesario. A esto se le llama cache snooping.

La memoria cache fue implementada desde los procesadores Pentium de Intel. A este tipo de cache se le conoce como L1-cache. Modelos recientes implementan varios niveles de cache (L2, L3, etc) donde cada nivel es mas lento que el anterior pero mas rápido que la memoria RAM. El manejo de los distintos niveles es dejado a nivel de hardware por lo que sistemas operativos como Linux asumen solo un nivel de cache.

Existe una segunda cache usada para acelerar el cálculo de las pysichal address llamado Translation Lookaside Buffer (TLB). Cuando una pysichal address es calculada, se utilizan la Page Tables la primera vez y el resultado se guarda en la TLB para que en posteriores accesos se utilize la dirección almacenada en la TLB en lugar de hacer todo el cálculo.

2.5 Pyshical Address Real World Example

En esta última sección se mostrará como Windows 8 en su versión de 64 bits almacena las linear address, llamadas virtual address en los sistemas operativos modernos, y hacer el cálculo de la physical address.

Debido a que windows mantiene los valores de las diferentes tablas en registros especiales nos puede mostrar el contenido y dirección de cada una de las cuatro tablas usadas. Utilizaremos el contenido asociado que nos muestra el kernel debugger de la última page table. Se deja como experimento personal al lector el comprobar las direcciones de las otras tablas. El valor que nos interesa es que son los bits del 12 al 20 del contenido de la page table, este valor coincide con el valor pfn mostrado por el kernel debugger. A este valor de concatenamos el offset de la virtual address que son los 12 bits menos significativos. Esto nos da .

Para comprobar que nuestro caculo es correcto, le pedimos al kernel debugger que ns muestre el contenido de la physical address caculada usando la instrucción `!dd`.